This is to certify that the

thesis entitled

VHDL Modeling Techniques and Design Verification

presented by

JIN-HYUNG LEE

has been accepted towards fulfillment
of the requirements for

M.S. degree in Computer Science

[illegible]
Major professor

Date April 25, 1990

DATE DUE	DATE DUE	DATE DUE
~~MAR 0 8 1993~~ 68~~187 1134~~		
DEC 1 7 1993		
FEB 3 1996 48 99		

VHDL MODELING TECHNIQUES AND DESIGN VERIFICATION

By

JIN-HYUNG LEE

A THESIS

submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Computer Science

1990

ABSTRACT

VHDL MODELING TECHNIQUES AND DESIGN VERIFICATION

By

JIN-HYUNG LEE

A VLSI system can be modeled and simulated for its verification of the function by using VHDL. In this thesis, some useful modeling techniques in VHDL have been presented for the efficient modeling of VLSI systems. VHDL semantics for process statements and functions are described, and delay characteristics are discussed. Design verification of VHDL descriptions is considered. In combinational circuits, we can verify the equivalence between the structural description and behavioral description by using timing tolerance and assertion statements. A clock adjustment scheme is introduced to verify sequential circuits. A strategy for treating general timing faults with VHDL is required for the purpose of exact design verification. An improved delay modeling scheme is described for the analysis and modeling of timing behavior. To detect timing faults, binary logic in VHDL is extended to use multiple valued logic. Detection procedures for timing faults are described with VHDL. As a benchmark test for the VHDL modeling techniques, a bit sync filter chip is chosen and modeled.

To My Parents for Their Love and Support

ACKNOWLEDGMENT

I wish to express my gratitude and appreciation to my major professor, Dr. Moon Jung Chung, for his guidance and encouragement during preparation of my thesis.

I would also like to thank to Dr. Anthony Wojcik and Dr. Dick Reid for participating as members of my graduate committee. Thanks are extended to ITT Aerospace/Optical for the financial support to carry out my research.

My sincerest thanks goes to my father and mother for their support and endless love for my life. Finally, for their patience, encouragement and support, I must express my heartful thanks to my wife, Min-Sun, and son, Kyung-Joon.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1. Introduction

VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL) is a language that can be used to express the function and logical organization of circuits, ranging from simple logic gates to complex digital systems [24,27]. We can model the behavior of the system and simulate it for design verification. Modeling in VHDL involves specifying the inputs and outputs of a system or device, and also describing its behavior (by specifying the output values as functions of the input values) and structure (in terms of interconnected subcomponents). VHDL allows a description to incorporate other design descriptions into units [27].

With some useful modeling techniques, VLSI systems can be modeled efficiently with VHDL. Analysis of VHDL semantics is necessary to get a precise model. Some critical features in VHDL modeling are selected for detailed investigation (process statements, functions, and delay characteristics). Semantics on the process statement provide an interesting feature for system modeling. A logic block can be modeled easily using this process statement and accompanying wait statement for the flow control of a VHDL program.

In VHDL, functions define algorithms for computing values or exhibiting behavior. One of the useful functions is the bus resolution function which defines the resolution of output values driving a common output signal. Using such a bus resolution function, signal multiplexing can be modeled for selecting a signal among multiple drivers. Alternative approaches for signal multiplexing are suggested without the bus resolution function.

VHDL supports two different models of time delay (inertial delay and transport delay) for modeling switching circuits [26]. However, these models does not reflect the delay which might be associated with wiring between devices. The delay model in VHDL should provide an accurate view of the timing associated not only with the logic gate, but also with the delay associated with the input wiring to the device.
A naming convention is described in order to develop VHDL models conveniently and to document them properly. This convention has been used for VHDL modeling of the bit sync filter chip in the SINCGARS (SINgle Channel Ground/Air Radio System).

Design verification of logic systems becomes increasingly important in the VLSI design process as the complexity of VLSI chips increases. Design verification should be done at the initial stage of the design before implementation, because design modification costs too much at later stages. The issue of the design verification using a hardware description language is significant since more designs are carried out by first modeling their behaviors and then by modeling their structures in VHDL. However, it is very difficult to determine whether a structural description of a design entity is equivalent to its behavioral description, since behavioral modeling is much different from structural modeling. In logic circuits, the correct operation requires proper timing operation as well as proper logical behavior. In this thesis, design verification strategies are proposed to determine equivalence between two models in VHDL descriptions by analyzing their timing behavior.

A design is usually verified using simulation, and such a design verification often requires comparing the simulation result with the design specification. In VHDL, the design specification is usually described by the behavioral model and is compared with the structural model for the verification purpose [2]. In order to verify equivalence in VHDL models, a method is suggested to examine timing tolerances in digital circuits. This verification method can be applied to combinational circuits, but has limitations in its applications to sequential circuits. A modified strategy is developed, which includes a new modeling technique in the behavioral description and a clock adjustment procedure for sequential circuits. All of these strategies involve the assertion statement in VHDL, which checks if a specified condition is true or not [25].

Design verification involving timing faults is complicated due to the additional efforts of detecting them and comparing each model. If a timing fault involves a spike of small duration, a *preprocessing module* could eliminate it, and design verification can be obtained. If, however, timing faults involve other problems such as hazards or races, it is not simple to verify them. Such problems should be detected and removed during the VHDL modeling process. A detection strategy for general timing faults with VHDL is required for the purpose of exact design verification.

In order to detect the abnormal behavior of timing faults, the gap between the simulation models and the physical circuits should be reduced [21], i.e. the precise delay model and additional features in VHDL is required in the analysis and modeling of these transient behaviors. An approach to the detection of timing faults is presented for both combinational and sequential circuits using VHDL. First, an improved delay modeling scheme is described for the purpose of modeling real situations. Next, the detection procedure for timing faults is presented based on a ternary logic scheme [8]. Binary logic in VHDL is extended to use multiple valued logic. Detection procedures for timing faults are described with VHDL.

Chapter 2 describes a survey of VHDL characteristics, and in Chapter 3 some modeling techniques in VHDL are presented. Chapter 4 describes design verification strategies in VHDL. In Chapter 5, an improved delay modeling scheme and timing fault detection method involving ternary logic are suggested. As a benchmark test for the VHDL modeling techniques, a bit sync filter chip, which is an element in SINCGARS, is chosen and modeled in VHDL. VHDL modeling report for the bit sync filter chip is included in Appendix 1. Appendix 2 has some examples for transient analysis with VHDL.

Chapter 2. VHDL Characteristics

The collection of information describing a particular piece of hardware is called *design entity*, which is the primary abstraction element in VHDL. It represents a portion of a hardware design such as a basic cell, chip, board, or system that performs a well defined function [26]. Each design entity is described in two parts; the definition of the interface between the entity and the outside world, called the *entity declaration*, and a design for the function and input/output transformation itself, called the *architecture body*. The entity declaration describes the design entity's ports, and contains any other information which is common to all bodies [27]. An example of an entity declaration, which is the full adder, is shown in Figure 2.1.

```
entity Full_ADDER
        (X, Y, Cin : in BIT ;
         Cout, Sum : out BIT) is
end FULL_ADDER
```

Figure 2.1 Entity Declaration in VHDL

The port and local item declared in the entity declaration are made available to all the bodies associated with this entity. The ports are the signals through which the design entity communicates with the outside world and it must be declared in the entity declaration part. In component instantiation of a block, a formal port of a component may be associated with a port of an enclosing entity. Generics provide a channel for static information to be communicated to a design entity from its environment, and it may be used to specify the timing characteristics of a entity, the size of ports, the number of subcomponents within a entity, or even the physical characteristics of a design such as temperature, capacitance, location, etc. Among these information of generics, the timing characteristics is the most useful for the VHDL modeling.

VHDL supports three distinct styles for system modeling and the description of hardware architectures. The first of these is *structural description*, in which the architecture is expressed as a hierarchical arrangement of interconnected components [26]. The second is *behavioral description*, in which the system model is described in sequential program statements that look like high level programming language. The last is *data-flow description*, in which the architecture is broken down into a set of concurrent register assignments and each assignment may be under the control of gating signals. Data flow description implies the style of description embodied in register transfer languages. The structural style of description displays the decomposition of a device into components and emphasizes the connections to be made among the components. In contrast, the data flow style emphasizes the flow of information between memory and gating elements. This flow is supervised and directed by control elements that are logically separate from the data paths. All three styles may be intermixed in a single architecture description.

A design entity is described in terms of interconnected components. Each instance of a component represents a portion of the design that may in turn be described by a lower level design entity built of interconnected components. In this manner, a hierarchy of design entities that represents the complete design can be constructed. A component is an abstract functional unit and it may represent a structural partitioning of the design or a functional decomposition of a large system. A designer can choose necessary components to build his own system. By using a behavior style of VHDL, a digital system can be described and modeled with a knowledge for what a device does (its function) without specifying how it does (its structure). For example, if a designer may wish to specify the behavior of a subsystem and leave the implementation details to others, he can model only the function of the system, which is not implementation technology dependent, by using a behavioral style of description. Implementation at the level of structural description might be performed by any other modeler.

Two important principles determine the course of simulation in VHDL and make VHDL suitable for the modeling of a physical system. First, *cause* always follows *effect*; a change in a signal value causes the execution of signal assignments that effect changes in the values of their targets. These effects may, in turn, cause additional changes, so that a *sequence* of events results. Many independent sequences of events can occur simultaneously. Second, changes can be made to take effect after a certain amount of *delay*. These principles would appear in a signal assignment statement in VHDL. A typical signal assignment consists of a driver and a target. A *driver* of a signal contains a current value and a waveform representing projected future values. Waveform elements are appended to a driver whenever a signal assignment is executed. Each element is stored there until its designated time arrives; it then becomes the current value of the driver. A driver is a source of the value of a signal. A signal may have multiple sources. If a signal has more than one source then all the sources participate in the calculation of the value. Such a signal must be a resolved signal, and the resolution function calculates one effective value from an array of some values. The *target* of a signal assignment is the signal or aggregate on the left hand side of the assignment operator. The simulator creates a driver for each element of the target of every concurrent signal assignment [25].

There are two kinds of statements in VHDL, which are sequential and concurrent statements. Sequential statements are used to define algorithms for the execution of a subprogram or process; they execute in the order in which they appear. But concurrent statements are used to define interconnected blocks and processes that jointly describe the overall behavior or structure of a design. Concurrent statements execute asynchronously with respect to each other [20]. The primary concurrent statement is the block statement, which groups together other concurrent statements defining an internal block representing a portion of a design. The other concurrent statement is the process statement, which represents a single independent sequential process representing the behavior of some portion of the design. Within a given simulation cycle, an implementation may execute concurrent statements in parallel or in some order. The language does not define the order, if any, in which such statements will be executed. A description that depends on a particular order of execution of concurrent statement is erroneous.

Design entities are stored in a file system called the VHDL Design Library (VLS). The entities in the design library represent self-contained functions, like a collection of hardware components. A *design library* is an implementation dependent storage facility for previously analyzed design units. A given implementation may have any number of design libraries. A library clause defines logical names for design libraries in the host environment. A working library is the one into which the library unit resulting from the analysis of a design unit is placed. Only one library might be the working library during the analysis of any given design unit [25].

Chapter 3. Modeling Techniques in VHDL

3.1 Process Modeling

3.1.1 VHDL Execution in the Process

A process statement defines an independent sequential statement representing the behavior of some portion of the design [25]. In VHDL, a process defines a sequential behavior of a design entity. The execution of a process consists of the repetitive execution of its sequence of the independent sequential statements. After the last statement in the sequence of statements of a process statement is executed, execution will immediately continue with the first statement in the sequence of statements.

If a sensitivity list appears following the reserved word "process", then the process statement is assumed to contain an implicit wait statement as the last statement of the process statement part. A process with a sensitivity list always waits at the end of its statement part, and event on a signal named in the sensitivity list will cause such a process to execute from the beginning of its statement part down to the end, where it will wait again. Such a process executes once at the beginning of simulation (at time 0) and is suspended at the end when it executes the implicit statements. But a process without a sensitivity list has an explicit wait statement for the flow control of a statement. We can model the logic block easily using this wait statement.

1. Signal and variable assignment in a process

A signal assignment in a process takes one Δ time unit, in other words, if the in-

put changes the value in the signal assignment, then the effect of a signal assignment occurs after Δ time unit [6]. However, a variable assignment does not consume any length of time, and the effect of input changes occurs immediately at the output. Figure 3.1 shows the difference between signal and variable assignment in a process. X1 and X2 are input signals and F is an output signal.

```
architecture PROC0 of CL0 is          architecture PROC2 of CL0 is
 signal T1 : bit := '0';               begin
 begin                                  process (X1, X2)
 process (X1, X2)                         variable T1 : bit;
  begin                                 begin
   T1 <= not X1 ;                         T1 := not X1 ;
   F  <= T1 or X2;                        F  <= T1 or X2 ;
 end process;                          end process;
end PROC1;                            end PROC2;
          (a)                                  (b)
```

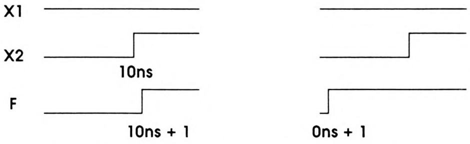

Figure 3.1 Signal and variable assignment in a process

When X2 goes to 1 from 0 at 10ns, output signal F in the program (a) rises to 1 at 10ns + 1Δ, but F in the program (b) goes to 1 at +1Δ. The reason is as follows: In program (a), the process executes once at time 0 ns, and that results will happen at time +1 Δ time unit. At 10 ns signal X2 rises to 1, then a process will be activated and signal F goes to 1. Program (b) shows when a variable is assigned in a process statement. Variable assignment does not take any length of time, and it is assigned at time 0. Then the effect of variable assignment occurs at time +1 Δ time unit since the first process execution is done at time 0 ns by default.

2. Preemption in a process

Since some statements in a process execute repetitively in sequential order, the second signal assignment preempts the first signal assignment when there are two different signal assignments for a signal in a process [11]. Same thing happens when two different values are assigned to one variable. Example programs for two signal assignments are shown below in Figure 3.2. X1 and X2 are input signals and F is an output signal for a program. A signal T1 is preempted at the process.

```
architecture PROC1 of CL0 is
    signal T1 : bit := '0';
  begin
   process (X1, X2)
     begin
       T1 <= X1 ;
       T1 <= not X1;
       F <= T1 or X2;
   end process;
end PROC1;
```

Figure 3.2 Signal preemption in a process

3.1.2 Timing Semantics of Process

In VHDL, a process statement calculates its output by using the value from the previous invocation process, not by using the current changed value. This characteristics of process statement is interesting and different from that of the other programming languages. Some examples make it easy to understand the timing semantics of a process.

One example would be half adder. The combinational circuit is usually modeled

by the block statement. However, if we use a process statement instead of block statement in VHDL modeling of the half adder, it causes a problem. Because a process statement is a repetitive sequential statement using the value from a previous invocation process, as described before, it generates incorrect results. In Figure 3.3: half-adder, Ainb and Binb change their values when the input Ain and Bin are entered. Sum21 and Cout change their value also according to changes in S1 and S2. However, the value of Sum2 signal assignment statement comes from the result of the previously invoked process, not from the changed value of present process. That is, when S1 or S2 changes its state, the value of Sum2 is not changed and it keep its previous value. Sum2 value doesn't go to '1' at 20ns+2 and it does not go to '0' at 30ns+2 either. Thus we should use a block statement instead of a process statement, which might result in incorrect output in such a combinational circuit. The process statement is not usually suitable for modeling of the combinational circuit, which consists of many sequential statements.

```
architecture proc_imp of half_adder is
    signal Ainb, Binb, S1, S2 : bit := '0';
 begin
   Gate_imp: process (Ain, Bin)
      begin
          Ainb <= not Ain ;
          Binb <= not Bin ;
          S1   <= Ainb and Bin ;
          S2   <= Ain and Binb ;
          Sum2  <= S1 or S2 ;
          Cout2 <= Ain and Bin ;
  end process;
end proc_imp ;
```

```
    TIME    |-------------------SIGNAL NAMES--------------------|
            |
    (FS)    |     A       B       S       C       S       S
            |     I       I       U       O       1       2
            |     N       N       M       U
            |                             T
            |
           0 |    '0'     '0'     '0'     '0'     '0'     '0'
          +1 |    ***     ***     ***     ***     ***     ***
    20000000 |    ***     '1'     ***     ***     ***     ***
          +1 |    ***     ***     ***     ***     '1'     ***
    30000000 |    ***     '0'     ***     ***     ***     ***
          +1 |    ***     ***     '1'     ***     '0'     ***
    40000000 |    '1'     ***     ***     ***     ***     ***
          +1 |    ***     ***     '0'     ***     ***     '1'
    50000000 |    ***     '1'     ***     ***     ***     ***
          +1 |    ***     ***     '1'     '1'     ***     ***
```

Figure 3.3 Structural description for a Half Adder

Another example is a frequency divider. The frequency divider can be well modeled by using a process statement. In Figure 3.4, which is a behavioral description of a divide-by-4 circuit, the process is invoked whenever clk_in is changed. If the input clock has just made a 0 to 1 transition, the value of f-count is increased. The initial value of f-count is 0, and it increases to 1 after the execution of this statement. When the next statement is executed, the value of f-count is still 0 because of the semantics of the process statement, that is, it gets the value from the previous invoked process. When this block counts 4 times, f-count is set to 0 and a output pulse is generated. Therefore, the signal f_count, in if clause, should be 3 not 4 in order to count 4 times, since a process statement defines an independent sequential statement [28].

```
architecture top04 of divide_by_four is
    signal f_count: integer := 0 ;
 begin
   process (clk_in)
     begin
        if (clk_in = '1' and not clk_in'stable) then
            f_count <= f_count + 1 ;
            if f_count = 3 then
              f_count <= 0 ;
              by_four_out <= '1' , '0' after clk_del;
            end if;
          end if;
      end process;
    end top04;
```

Figure 3.4 Behavioral description for frequency divider

3.1.3 Process Interaction in Multiple processes

When there are two or more process statements in a VHDL program, each process statement executes concurrently. If there is more than one signal assignment statement for a signal in a process, a following assignment preempts the previous one. However, if there is more than one process in a VHDL model, they execute in a parallel fashion as shown in following example. (X1 and X2 are input signals and F is an output signal)

```
architecture PROC3 of CL0 is
  signal T1 : bit := '0';
  begin
    process (X1)
      begin
        T1 <= not X1 ;
    end process;
    process (T1, X2)
      begin
        F  <= T1 or X2;
    end process;
end PROC3;
```

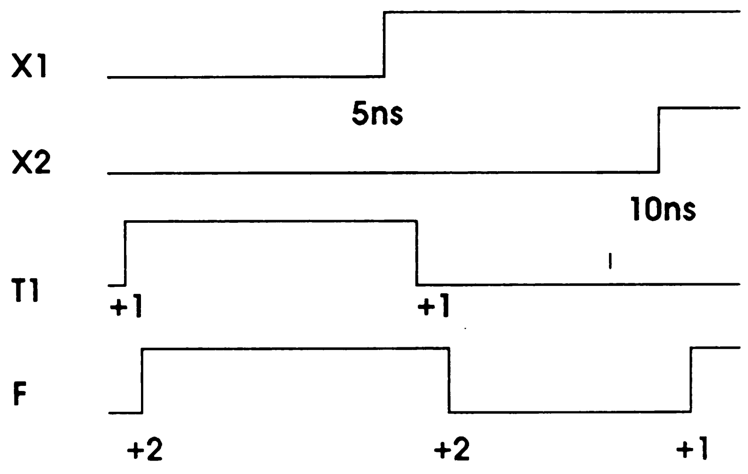

Figure 3.5 Multiple process statements in a VHDL program

1. Signal and variable assignment in multiple processes.

If a signal is declared in the declaration part of an architecture body, each process shares the value of that signal, and they execute concurrently. Since a signal T1 is a global signal between two processes, the effect of changing value occurs in the second process after Δ time unit in Figure 3.5. The same thing occurs in case of the global variable assignment.

Suppose there is not any shared signal between two processes, and those processes include the same variable. Then each variable is a separate one to its own process and independent to other process. Each process executes independently with each variable.

2. Intercommunication between multiple processes

In order for one process to communicate to other process, there should be shared variables or signals. Without shared variables or signals the intercommunication between two processes cannot be made, since each variable to each process is independent to the other. By using characteristics of the process statements and the intercommunication among multiple processes, many useful circuit blocks can be modeled. As an example, consider a clock generator in Figure 3.6. Here, the period of a clock becomes different depending on the value of *cntl_clk*, and then it generates two different clock periods.

```
architecture hilow4 of Period_clock4 is
    signal cntl_clk , clkenb : bit := '0' ;
  begin
    t_clock: process
            begin
               wait for 23400 ps;
                 clkenb  <= '1' ;
               wait for 514800 ps;
                 cntl_clk <= '1' ;
            end process ;

    main_clk: process (clkenb, cntl_clk, clk_in)
             begin
                if (clkenb = '1') then
                   if ( cntl_clk = '0' ) then
                      if (clk_in = '1') then clk_in <= '0' after high_time1;
                      elsif (clk_in = '0') then clk_in <= '1' after low_time1;
                      end if;
                   elsif ( cntl_clk = '1' ) then
                      if (clk_in = '1') then clk_in <= '0' after high_time2;
                      elsif (clk_in = '0') then clk_in <= '1' after low_time2;
                      end if;
                   end if;
                elsif (clkenb = '0') then clk_in <= '0' ;
                end if;
             end process;
end hilow4;
```

Figure 3.6 VHDL model for clock generator using multi-process

The implementation of this clock is based on multiple process statements. It is modeled by two processes, in which one is for generating the control signal and the other is for main clock generating routine, depending on the results from first process. In this circuit, it generates the clock whose period is high_time1 plus low_time1 before 514800 ps, and high_time2 plus low_time2 after 514800 ps, and it does not generate any clock during the initial phase before 23400 ps.

3. Same signal assignment in multiple processes

If there is a shared signal among multiple processes and each process has different signal assignment statements in a VHDL program, this program generates error since there are multiple sources for this signal that are not resolved [28]. Each process executes concurrently and tries to assign two values to the same signal. In comparison to Figure 3.2, which has two different signal assignment in a process, that program executes in sequential order and the second signal preempts the first signal. However, same signal assignments in multiple processes do not preempt and this signal should be resolved. These signals need a bus resolution function and an example program is shown in Figure 3.7.

```
architecture PROC5 of CL1 is
      signal T1 : bit := '0';
    begin
    process (X1, X2, T1)
       begin
          T1 <= not X1 ;
          G <= T1 or X2;
    end process;
    process (X1, X2, T1)
       begin
          T1 <= X1 after 1ns;
          F  <= T1 or X2 after 2ns;
    end process;
end PROC5;
```

Figure 3.7 A VHDL program for multiple processes

3.2 Functions and Signal Multiplexing

3.2.1 Functions and Packages in VHDL

There are two forms of subprograms in VHDL: procedures and functions. A procedure call is a statement and a function is an expression and returns a value. In VHDL, functions defines algorithms for computing values or exhibiting behavior. They may be used as computational resources to convert between values of different types and to define the resolution of output values driving a common signal or to define portions of a process.

The definition of a function can be given in two parts: a function declaration defining its calling conventions, and a function body defining its execution [25]. All functions should be defined within a package. Packages provide a means of defining these and other resources in a way that allows different design units to share the same declarations. Function name, parameter list, and return value must be declared in the package declaration part, and the bodies of any functions declared in the package declaration can be contained in a package body. Any package declaration must be model-generated before its corresponding package body can be model-generated. An example program for the bus resolution function is shown in Figure 3.8 and 3.9.

```
package  TOOLS2 is
  type MVL_VECTOR is array (NATURAL range <> ) of TRISTATE;
  function TRISTATE_RESOLUTION ( INPUT : MVL_VECTOR )
  return  TRISTATE;
end TOOLS2;

package body TOOLS2 is
```

```
    function TRISTATE_RESOLUTION ( INPUT : MVL_VECTOR )
    return  TRISTATE is
        variable RESOLVED_VALUE : TRISTATE ;
    begin
      for I in INPUT'left to INPUT'right loop
         if INPUT (I)  /= 'Z' then
            RESOLVED_VALUE := INPUT (I);
            exit;
         end if;
      end loop;
      return RESOLVED_VALUE;
     end TRISTATE_RESOLUTION;
  end TOOLS2;
```

Figure 3.8 Bus resolution function (1)

3.2.2 Signal Multiplexing

When we model the VLSI circuits, signal multiplexing is necessary in many cases. Approaches in signal multiplexing with VHDL can be divided largely into two types: *with* or *without* the bus resolution function.

Bus resolution function is used to resolve the values of the drivers into the values of the signal. It must have previously been declared, and is called whenever the signal is referenced [25]. Bus resolution function is application dependent and modeler should define his own function for each purpose. Note that the bus resolution function in Figure 3.8 scans the drivers of INPUT and selects the value of the first driver that is not equal to Z, the assumption being that only one of the drivers is active. However, this assumption that only one driver is active is certainly not true in all modeling situations. Some bus lines are required to carry a signal that represents the logical OR of

the outputs of several bus drivers that are connected along the bus, which is the property of the WIRED-OR function. Such a WIRED-OR resolution function is in Figure 3.9.

```
package  busr2 is
  function RESOLVE_BIT (DRIVERS : bit_vector) return bit ;
end busr2;
package body busr2 is
   function RESOLVE_BIT (DRIVERS : bit_vector) return bit is
       variable COUNT : integer;
       variable VALUE : bit;
   begin
      VALUE := '0';
      for COUNT in DRIVERS'LOW to DRIVERS'HIGH loop
         if DRIVERS(COUNT) = '1' then
             VALUE := '1';
         end if;
      end loop;
      return VALUE;
   end resolve_bit;
end busr2;
```

Figure 3.9 Bus resolution function (2)

1. Guarded Signals

Let us model the multiplexer, which selects one of the input signals according to the control signal. Assume that, in the logic to be modeled, the value of the control signal is mutually exclusive in a logic block. The intent of the model is that when a signal *cntl* is '1' data1 will be assigned to S, while when *cntl* is '0' data2 will be assigned to S [6]. We can get a model in VHDL as follows :

```
architecture top_mux1 of mux1 is
    signal S: RESOLVE_BIT bit register (bus);
  begin
     L1: block (cntl = '1')
         begin
            S <= guarded data1 ;
         end block;
     L2: block (cntl = '0')
         begin
            S <= guarded data2 ;
         end block;
         mux1_out <= S ;
end top_mux1;
```

Figure 3.10 Multiplexer Implementation in VHDL

To model the problem of the bus resolution combined with guarded signal assignment statements, VHDL provides a special mechanism for signals used in guarded signal assignments as in the above example. The block whose guard is TRUE will have its value coming through the bus resolution function [26]. A guarded signal is assigned to the values under the control of Boolean-valued guard expressions (or guards). Whenever a guarded signal designated as being a register or a bus has its block guard become FALSE, its driver is assumed to be off, i.e. it is ignored by the bus resolution function. When a given guard becomes false, the drivers of the corresponding guarded signals are implicitly assigned a null transaction to cause those drivers to turn off. For the case where all the block guards are off, if S is of kind *register* it will retain its last value, while if S is of kind *bus* it will be assigned the default value provided by the bus resolution function.

These representations "register" and "bus" are only used in VHDL with guarded signals. A signal which is multiple driven, requires a bus resolution function. If, however, it is not the object of a guarded signal assignment, it cannot be declared to be a kind of bus or register.

2. Alternative Approaches

There are several alternative implementations for multiplexers. Consider the following simple example: The intent of the two processes is to have a signal S got K20khz when freq_cntl = 0 and K16khz when freq_cntl = 1. This is a signal multiplexing problem again, i.e. a simple bus resolution function cannot model this behavior. One of the feasible solutions is to use a simple *when statement* as follows: In this implementation, Mux1_out signal will be multiplexed depending on the value of freq_cntl.

```
architecture logic1 of mux1 is
  begin
   A: block
      begin
        Mux1_Out <= K20KHZ after 1ns when freq_cntl = '0' else
                    K16KHZ after 1ns                       ;
      end block;
end logic1;
```

Another alternative is to use a process statement as follows: We can get multiplexed signals by using the process statement instead of the block statement. In some cases, the process statement is a lot easier than the block statement in VHDL modeling, but modelers should be cautious in selecting between a process and a block statement [11].

```
architecture logic1 of mux1 is
  begin
```

```
    A: process (freq_cntl)
       begin
         if ( freq_cntl = '0' ) then
              Mux1_Out <= K20KHZ after 1ns ;
         elsif ( freq_cntl = '1' ) then
              Mux1_out <= K16KHZ after 1ns ;
         end if;
       end process;
end logic1;
```

A third alternative is to use a process and when statement. A similar signal multiplexing problem can be represented with processes [1]. Processes A and B assign values to distinct signals S1 and S2 which are then multiplexed. This solution defines a multiplexer that is external to the processes. In this example, both signals of S1 and S2 cannot be unstable since two conditions of freq_cntl = 0 and 1 are exclusive to each other.

```
A:  process(freq_cntl)
    begin
    if (freq_cntl = '0') then S1 <= K20khz after 1ns;
    end process;

B:  process(freq_cntl)
    begin
    if (freq_cntl = '1') then S2 <= K16khz after 1ns;
    end process;

  S <= S1 when not S1'stable else
       S2 when not S2'stable else
       S;
```

All of above solutions do not use the bus resolution function and define a multiplexer. These implementations make it easier to understand the VHDL modeling in the level of the behavioral description. Signal multiplexing with guarded blocks and processes shows interesting results and one can make good models on some logic circuits for the signal multiplexing.

3.3 Delay Characteristics

3.3.1 Delay Models in VHDL

VHDL supports two different models of time delay that correspond to the different kind of delays encountered in real world. The *inertial Delay* is a delay model for modeling switching circuits [22,26]. This models the time lag between stable inputs and valid outputs of a semiconductor logic element. The output will appear after the input has persisted for the indicated delay time, but no change in the output will occur if the input changes persist for less than the delay time. In other words, a pulse whose duration is shorter than the switching time of the circuit will not be transmitted. Thus, the inertial delay model describes the required persistence of inputs and the throughput delay of an element with a single time value.

Because the inertial delay is so common in the real world, it has been adopted as the default delay model in VHDL. All of the signal assignment we have used so far in this thesis exhibit inertial delay. There are, however, some delays in the real world that cannot be modeled in this way. For example, the wave propagation delay through a metal conducting path. If the path can be modeled as a pure resistance, then any change on the input, no matter how brief, will be reflected on the output after the propagation delay. The *transport delay* is a characteristic of hardware devices (such as transmission line) that exhibits nearly infinite frequency response. This model removes the requirement for the minimum duration for input pulse. Any changes on the input can cause an output change and any pulse is transmitted, no matter how brief the input transient and regardless of the delay [27].

Consider following VHDL program, to clarify between the inertial delay and transport delay model.

```
architecture delay_inert of and_test is
 begin
  L1: block
      begin
          AND_OUT <= A and B after 2ns;
      end block ;
end delay_inert ;

architecture arc_and of tand is
    component c_and_test
          port( a , b : in bit; and_out : out bit );
    end component;
    signal A, B, AND_OUT : bit := '0';
    for L_and_test : c_and_test use entity and_test (delay_inert);
begin
    A  <= '1' after 3 ns, '0' after 6 ns;
    B  <= '1' after 5 ns, '0' after 8 ns;
    L_and_test : c_and_test  port map (a, b, and_out );
end arc_and ;
```

Figure 3.11 A VHDL program for the inertial delay model

The above program does not generate an output signal on the output AND_OUT, since the duration of the output for AND operation is only 1ns, which is less than minimum duration for a signal assignment AND_OUT (2ns). The inertial delay model checks only after completion of the logical operation. If this signal assignment is based on the transport delay model, then the output for duration of 1ns will appear at 6ns in AND_OUT.

VHDL has a third delay model called *delta delay*. This is a special delay model for modeling the sequencing of events, without considering the actual times at which they occur. It also can be used to perform unit delay simulation, since each delta delay between events can be interpreted as taking the same amount of time as every other delta delay. Delta delay represents an infinitesimal delay, less than any measurable time (i.e. femto seconds), but still larger than zero. A value assigned with delta delay will take effect in the future. However, because the delay is infinitesimal, the value will take effect before any values assigned with any real time delay, no matter how small. Also, any number of consecutive delta delays will never add up to any real time.

3.3.2 Wire Delay

Previous models in this section have anticipated only the propagation delay associated with the device. These models did not reflect the delay which might be associated with wiring between devices. In ASIC design especially, the assumption of no wiring delay can be dangerous since many designs of this type might be important if timing effect from wires is dominant [7].

For example, consider an example in following Figure 3.12.

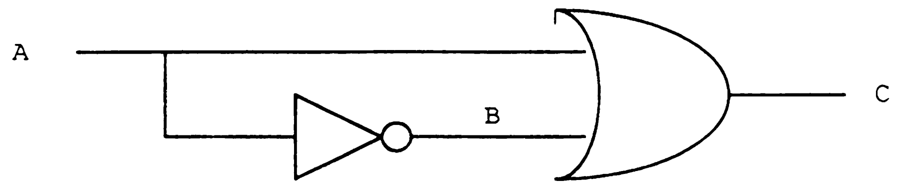

Figure 3.12 A Circuit representing Wire Delay

We determine if there is hazard or not in above circuit. If there is no information about the wire delay, it almost impossible to determine a hazard in a real circuit [4]. For example, when input A changes its state 0 to 1 output C can generate two different outputs, depending on the length of wire delay as well as the delay of the inverter. If the wire delay is larger than the inverter delay then a hazard occurs at output C. However, if we assume zero delay through all wires and a unit delay through gates then a hazard does not occur. Similarly, when input A changes its state 1 to 0 the hazard can be observed at output C depending on wire delay. If the wire delay is much larger than the inverter delay then a hazard does not occur. However, if we assume zero delay on the wire and unit delay on gates as is the case in VHDL, then a hazard occurs.

Thus we need information about the wire delay as well as gate delay in order to determine if there is a hazard or not, and the concept of the wire delay is very important in the real circuit modeling. The delay model in VHDL should provide an accurate view of the timing associated not only with the logic gate, but also with the delay associated with the input wiring to the device.

3.4 Documentations in VHDL

In order to develop VHDL models efficiently, uniform and consistent naming conventions should be applied [7]. In this thesis, a naming convention is proposed from my experience. Concerns are about the entity and architecture name, component name, and label name in component instantiation statements. An entity and architecture name are made according to their characteristics. For example, entity name is divider_prgm and architecture name is pulse_sw for modeling of the programmable divider, which is a pulse swallower, in appendix A: VHDL modeling for bit sync filter [6,13]. However, divide-by-4 and divide-by-5 circuits are not pulse swallowers but regular frequency dividers, and then their architecture names are chosen to regular_4 and regular_5 respectively. For a component name, only prefix C is added to an entity name for simplicity reasons. Similarly, prefix L is added to the entity name for the label name in the component instantiation statement. The name of the generic constant and the port should be reasonable and contain information which is associated with its meaning. These naming conventions will improve readability and consistency of VHDL models. Figure 3.13 shows an example for a naming convention used in the modeling of a bit sync filter chip.

```
for L_divider_prgm: c_divider_prgm  use entity divider_prgm(pulse_sw);
for L_divider_41, L_divider_42:
                    c_divider_4     use entity divider_4(regular_4);
for L_divider5    : c_divider_5     use entity divider_5(regular_5);
for L_mux1        : c_mux1          use entity mux1(simple1);
for L_in_shot     : c_in_shot       use entity in_shot(fall_edge_det);
for L_mid_shot    : c_mid_shot      use entity mid_shot(rise_edge_det);
for L_in_edge_det : c_in_edge_det   use entity in_edge_det(inin);
for L_mid_edge_det: c_mid_edge_det  use entity mid_edge_det(inmid);
for L_clk_enable  : c_clk_enable    use entity clk_enable(synch);
for L_phase_detect: c_phase_detect  use entity phase_detect(cntl_updn);
for L_up_down     : c_up_down       use entity counter(up_down);
for L_mode_select : c_mode_select   use entity mode_select(mode_cntl);
for L_mux2        : c_mux2          use entity mux2(simple2);
for L_final_shot  : c_final_shot    use entity final_shot(muxout);
for L_clk_adjust  : c_clk_adjust    use entity clk_adjust(p10_adjust);
```

Figure 3.13 An example for naming convention on VHDL modeling

When a large system is going to be modeled with VHDL, a good and proper documentation handling method should be required. For example, various versions of models will be generated during the development of the VHDL model for a system. There should be a big caution in this version control, and the behavioral description and structural description for the system must be well maintained. A good document control method will be a great help for the development of a VHDL model development and also a well defined software engineering technique is indispensible in project management of the VHDL modeling for the VLSI system.

Chapter 4. Design Verification with VHDL

In top down design of a circuit, it is necessary to verify whether a structural description of a design entity is equivalent to its behavioral description in VHDL models. However, it has been not easy whether a structural model is equivalent to its behavioral model. Consider an example for this problem - a half adder, which is shown in Figure 4.1.

```
architecture BEHA_HA of HA is
 begin
   process (Ain, Bin)
       begin
          if (Ain = Bin)  then       Sum1 <= '0' ;
                       if (Ain = '1')  then  Cout1 <= '1' ;
                       end if;
          else         Sum1 <= '1' ;
                       Cout1 <= '0' ;
          end if;
   end process;
end BEHA_HA ;
architecture GATE_HA of HA is
   signal Ainb, Binb, S1, S2 : bit := '0' ;
 begin
   block begin
       S1   <= (not Ain) and Bin ;
       S2   <= Ain and (not Bin) ;
       Sum2  <= S1 or S2 ;
       Cout2 <= Ain and Bin ;
   end block;
end GATE_HA ;
```

Figure 4.1. : Design Verification Example (1) - A Half Adder

In the structural description GATE_HA of half adder HA, the outputs, Sum2, change their states at 2 Δ time, after input change of Ain and Bin. However, in the behavioral description BEHA_HA, the value of Sum1 is changed at 1 Δ time after input value change as shown in Figure 2.2. Thus outputs of two entities are not said to be equivalent if we compare them using the following assertion statement shown in Figure 4.3.

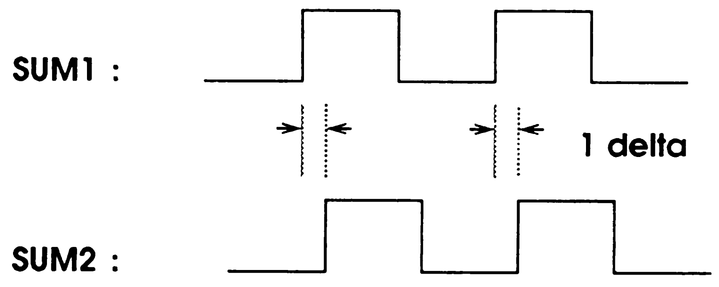

Figure 4.2. : Output waveform for behavioral and structural model

```
Assert ( Sum1 = Sum2 )
report  "Simulation result doesn't match" ;
```

Figure 4.3 Assertion in VHDL

A method of using timing tolerance is suggested to verify the equivalence between two descriptions in combinational circuits. This method is extended to the sequential circuits for their verification, and discusses the limitation of these methods.

4.1 Timing Tolerance in Combinational Circuits

A simple way for verifying the equivalence is inserting a delay on the final output for two descriptions, thus enforcing the equal delay for the behavioral and the structural descriptions. This approach can be applied to the previous example - half adder in Figure 4.1. In order to make up this timing difference between two outputs for the behavioral and the structural models, a delay can be put on the final output of both descriptions. The same delay - 1ns was inserted on the signal assignment statement of Sum and Cout for both descriptions, and then each Sum and Cout is changed at the same time. Now the assertion statement in Figure 4.3 will be satisfied.

However, this method cannot be applied when there are multiple paths from input to output with different delays (i.e. multiplexer), or when the delay of the gates depends on the input values (for example, rising/falling delay in a inverter). Also this approach has a limitation when the complexity of the gates are large.

A method, which is based on the timing tolerance between the behavioral and the structural models is suggested. In specification of hardware design, the gate delay in a circuit usually has a tolerance, and if the other particular design is within this tolerance we would say two circuits are equivalent. Timing tolerance can be determined in several ways. For a device, there is a manufacturing tolerance on the timing specifications which represents minimum and maximum value. The exact timing of a device depends on its environment, namely loading, temperature, and supply voltage. Timing tolerance might take the form of a maximum set up time or be defined by the designer.

In the previous example, suppose that the timing tolerance between two descriptions is *N* (generally it might be in fs, ps, or ns). The design verification can be obtained by using assertion statements as shown in following Figure 4.4 (here, *h* = *N/2*).

```
process (Sum1, Sum2)
   Assert ( not (not Sum1'stable(N) and not Sum2'stable(N))
           or (Sum1 = Sum2))
   report " Simulation result doesn't match ";
   Assert ( (not (not Sum1'stable(N) and Sum1'stable(N-h))
                  or (not Sum2'stable(N+h))) and
                 (not (not Sum2'stable(N) and Sum2'stable(N-h))
                  or (not Sum1'stable(N+h))))
        report " Simulation result doesn't match ";
      end process
```

Figure 4.4 Modified Assertion for Half Adder - *Assertion Process Shell*

The above process involving assertion statements checks whether Sum1 and Sum2 behave identically within timing tolerance (here N = 2ps). The first assertion checks whether Sum1 and Sum2 have the same value whenever both of them change within timing tolerance. However, there are some chances that only one of Sum1 or Sum2 has been stable while the other has been unstable, or both of Sum1 and Sum2 have been stable. The second assertion statement checks if Sum1 has been changed within the time between N and N-h, then Sum2 must be changed within the timing tolerance of Sum1 (between N+h and N-h). We call the above process as *Assertion Process Shell* for equivalence check, and this assertion process shell would be used for checking other models. Figure 4.5 shows the stability range for signal Sum2 when Sum 1 is not stable between -2ps and -1ps, in order not to violate second assertion statement.

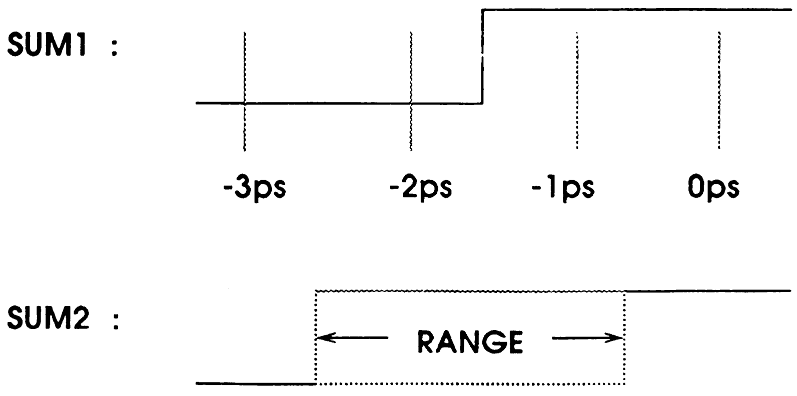

Figure 4.5 Stability Range for Sum1 and Sum2

Here, Sum2 should be unstable between -3ps and 0ps. The meaning of the second boolean expression of the second assertion is the opposite of the first one.

4.2 Verification in sequential circuits

Our method for design verification is working on most of combinational circuits, but it has a limitation in its application to sequential circuits. In many sequential circuits, outputs of circuits resulting from input changes may take more than one clock period. In this case, our previous method cannot be applied directly.

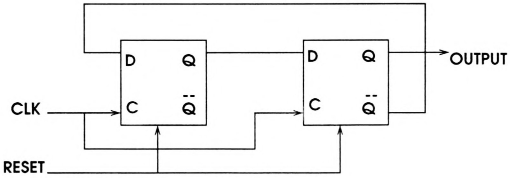

Figure 4.6 : Logic Diagram for Frequency Divider

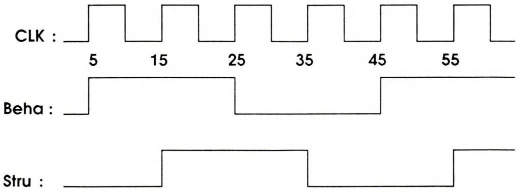

Figure 4.7 : Input / Output Waveform for Frequency Divider

Consider the frequency divider as an example. It is a divide_by_4 circuit (Fig. 4.6), which is a synchronous sequential circuit. Input/output waveform is shown in Figure 4.7. Figure 4.8 shows the behavioral description for the frequency divider, and the structural model is straightforward and is not given here.

```
architecture beha_desp of divide4_1 is
    signal count: integer := 0 ;
 begin
   process (clk_in)
     begin
        if  (clk_in = '1' and not clk_in'stable)  then
             count <= count + 1 ;
             if  count = 0  then by_four_out <= '1' ;
             elsif  count = 2  then by_four_out <= '0' ;
             elsif  count = 3  then count <= 0 ;
             end if;
        end if;
   end process;
end beha_desp;
```

Figure 4.8 : Design Verification Example (2) - Frequency Divider

The output from the behavioral description generates the first state at the first rising edge of the clock, but the output from the structural description generates its first state at the second rising edge of the clock, as shown in Figure 4.7. The behavioral model takes 1 Δ time unit, while the structural model takes 10ns + 2 Δ time unit, if the delay length in a D-FF is assumed to 1 Δ time (Δ represents a delta delay unit). The timing difference between two descriptions is 10 ns + 1 Δ time, which is longer than the clock period (10ns). Thus if we apply timing tolerance approach, a timing tolerance can be confused with the actual clock period. Therefore, our previous method cannot be applied to the circuit, in which timing tolerance is longer than the clock period.

A modified approach for the sequential circuit is proposed. In most of the sequential circuits, the output of circuits are generated after several clock periods. We can get the number of clock difference between two outputs from the behavioral and structural description after analysis of two models, for example, as shown in the following Figure 4.9:

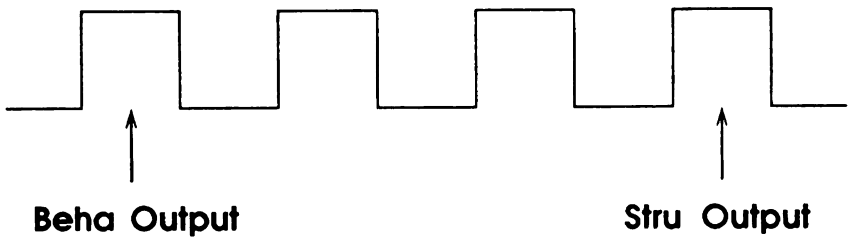

Figure 4.9 : Typical output waveform for behavioral and structural model

In order to make up this clock difference, the amount of clock difference is inserted into the behavioral model, so that both outputs for the structural and behavioral models are generated in the same clock period. After adjusting the clock difference, we can check the timing difference and Figure 4.9 can be modified to Figure 4.10. Now timing tolerance between two descriptions becomes within some number of delta time units.

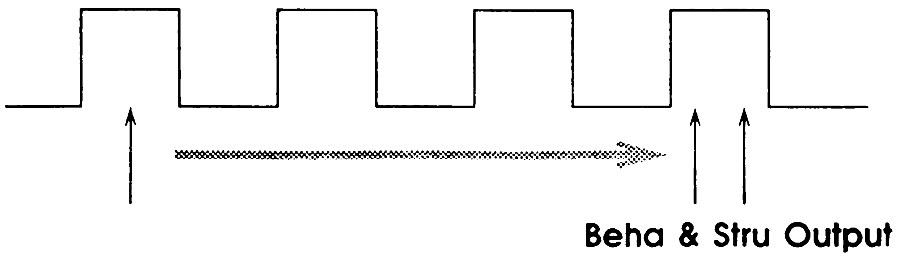

Figure 4.10 : Modified output waveform for the behavioral and structural models

In summary, design verification procedure for the sequential circuit is formally described as follows:

1) Get clock difference between the behavioral and structural models

2) Adjustment of clock difference

3) Verification by the timing tolerance

This approach is actually a combined method from the simple method and a method based on timing tolerance. In order to apply this modified approach, it is assumed to know how many clock periods are required for the real circuit operation. Let us apply this combined approach to the frequency divider. Its behavioral model generates the output at the first clock, but the structural model generates at the second clock, i.e. there is 1 clock difference between two outputs from these models. A delay of clock period (10 ns) is inserted on the behavioral model, so that both outputs can be generated during the same clock period, with difference of 1 Δ, which is within the timing tolerance.

This design verification procedure can be implemented actually in two ways. One way is that a generic constant - *clk_diff* is used for the clock difference in the behavioral model for the frequency divider. We do not have to change the behavioral model for each clock difference in this implementation. A new model is shown below.

```
architecture beha_desp of divide4_1 is
      signal count: integer := 0 ;
 begin
    process (clk_in)
      begin
         if (clk_in = '1' and not clk_in'stable) then
               count <= count + 1 ;
               if count = 0  then  by_four_out <= '1'  after clk_diff;
               elsif count = 2  then  by_four_out <= '0'  after clk_diff;
               elsif count = 3  then count <= 0 ;
               end if;
```

```
        end if;
    end process;
end beha_desp;
```

Figure 4.11 : Modified Model for the Frequency Divider

This model can be verified by using the *Assertion Process Shell* as shown in Figure 4.4, in which Sum1 and Sum2 is replaced by_four_out and Stru_out respectively. (Stru_out is the output for structural model).

The other way to implement the verification of a sequential circuit is by using the *verifier*, which is a top level VHDL description for the structural and behavioral models. The verifier performs the actual verification by adjusting the clock difference for the component of the behavioral model in the top level description. In this implementation, the behavioral model does not have to be modified and the original model is used for verification. For example, a verifier is needed in order to verify the frequency divider. Figure 4.12 shows a portion of the verifier.

```
Beha_out <= by_four_out after clk_diff;
L_beha_freq: c_beha_freq
            port map (clk_in, by_four_out);
L_stru_freq: C_stru_freq
            port map (CLK_IN, Stru_out);

-- Assertion Process Shell
   process (by_four_out, Stru_out)
```

Figure 4.12 A verifier for frequency divider

Here, c_beha_freq is the behavioral model and c_stru_freq is the structural model of the frequency divider respectively. The verifier includes the assertion process shell from Figure 4.4 and it would verify by the timing tolerance method.

4.3 Timing Faults in Verification

Verification between the behavioral and structural model in VHDL can be obtained by using our approach on both combinational and sequential circuits. If, however, there are any timing faults, i.e. spikes or hazards, in the combinational or sequential circuit, the circuit behavior becomes complicated. As a simple example, in a half adder, if the value of Ain and Bin change their state simultaneously (from 01 to 10), then 0-static hazard, which is that a circuit output goes to 1 when it should remain 0, is generated on the output Sum1 for the structural description. But there does not exist a hazard for the behavioral model of the half adder. Therefore, our verification methods cannot be applied directly here.

As an another example, consider following two VHDL models.

```
Model 1:  C1 <= A or (not B) ;
Model 2:  NB <= not B;
           C2 <= A or NB;
```

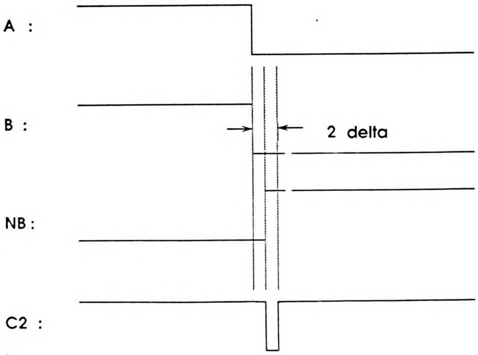

Figure 4.13 : Timing Diagram for Model1 and Model2

These two models describe the same circuit, but Model 2 uses an intermediate signal NB and Model 1 involves only one signal assignment. Figure 4.13 shows timing diagram for these models, and a spike can be observed at signal C2 in Model 2.

A spike does not occur in model 1, since *(not B)* does not take any length of time. However, a signal assignment for NB consumes one Δ in model 2. In this case, it is not easy to verify equivalence between two models due to a spike in model 2. A *preprocessing module* for design verification is needed to delete a spike for short duration. The preprocessing module is based on the inertial delay model, in which a pulse whose duration is shorter than the switching time of the circuit, will not be transmitted.

If an appropriate amount of inertial delay is provided, a spike could not proceed through the output gate. In order to perform such a model, an *improved* inertial delay model is necessary in the preprocessing module, which blocks timing faults which duration is for a very small amount of time. For example, suppose its duration is for some number of *delta time unit*. Then the timing fault can be blocked by the following statement.

```
A <= B after N delta ;
```

Here, N is a small number. If this model is available in VHDL, then the timing faults for N delta time do not occur in the output and a signal through the preprocessing module will be spike-free. (However, this model cannot be supported in present VHDL and it should be provided in the near future.) Now two models can be verified. A preprocessing module for timing faults check can be implemented in the VHDL as shown in the following Figure 4.14.

```
architecture delta_test of prep is
  begin
```

```
        block begin
            delta_out <= delta_in after N delta ;
        end block;
    end delta_test;
```

Figure 4.14 : A Preprocessing Module

Here, delta_in is a signal which might contain a timing fault, and delta_out is a spike-free signal respectively.

Chapter 5. VHDL Modeling for the Transient Analysis

In the previous chapter, a verification method for spikes has been described, which is a special case of timing faults. If, however, timing faults involve hazards and races, the circuit behavior becomes unpredictably complicated and it may cause critical situations in the digital circuits. Such hazard and race conditions should be detected and eliminated during the VHDL modeling process. In order to detect timing errors, the precise delay model and some additional features in the VHDL is required, which are useful for analysis and the modeling of these transient operations in digital circuit. An improved gate delay modeling for timing behavior and timing fault detection procedures with VHDL are described.

5.1 Delay Modeling for Timing Behavior

5.1.1 Survey of Delay Modeling

Several different approaches for delay modeling have been developed [14,21,22]. The first delay model implemented in the event-driven simulators is *a nominal delay model*. In that model, a single delay value is assigned to each kind of logic element. But some devices have different signal rise and fall times due to various electrical parameters such as its input parameter and load capacitances. Such devices can be modeled by assigning two delays of t_{phl} for transition from 1 to 0 and t_{plh} for transition from 0 to 1. This model is referred to *a rise/fall delay model*. In this model, if the output change caused by the first input change occurs later than that by the later input change, the events will be preempted, and the output would be assigned to be an unpredictable value or an error state. The more precise modeling of delay is called *a delay ambiguity model or a min/max delay model*. In this model, the logic circuits operate with a propagation delay somewhat between a minimum value t_{pdm} and a maximum value t_{pdM}. These delays define an ambiguity region of duration t_{pdm} - t_{pdM} and the gate signal changes the value sometime within this region. However, in this model, since the ambiguity region propagates through the elements in an additive fashion, the region at the output node widens compared to that at the input. Thus the model results in a worst-case behavior. Other methods proposed for the delay ambiguity modeling are the Monte Carlo simulation, where all combinations of delays can be considered and the delay distribution can be approximated by the Gaussian curve. To detect and model the unpredictable state due to spikes, hazards, and races, at least three value modeling is necessary, including 0, 1, X(unpredictable or unknown), since such abnormal states cannot be generated and the effect of these states cannot be propagated with the binary logic simulation model.

A gate is normally evaluated by two basic operations; a delay operation causing the signal delay and a functional operation to give the output value. There are two possible ways to set up the gate model [21]. The first model is the input-side delay model, which applies the delay operation to all of its input initially, and then performs the functional operation to yield the output. Thus,

$$\begin{aligned} Z &= F(D(x1,x2, \ldots ,xn)) \\ &= F(D1(x1), D2(x2), \ldots ,Dn(xn)) \end{aligned}$$

where (x1, x2, ..,xn) are the input signals and Z is the output. F is the functional operation and D is the delay operation of the gate, respectively.

The other model is the output-side delay model, which performs the functional operation first and then the delay operation follows, in contrast with the input-side delay model. Thus

$$Z = D(F(x1, x2, ..., xn))$$

These two models perform different behaviors when the input nodes change simultaneously. In the output-side delay model, the functional operation is performed first, and this results in the output of the gate being always correct. However, in the input-side delay model, the delay operation is performed first, which results go to the function unit. Since the functional operation is performed on the output of the delay operation, the unpredictable output, such as spikes, might be produced when the inertial delay model is adopted. Therefore, the output-side delay model is better than the input-side delay model in timing analysis. The delay model in VHDL (inertial and transport delay) is based on the output-side delay model.

5.1.2 Improved Delay Modeling for VHDL

The precise delay model for logic circuits is very important, in particular, for detecting timing faults, such as spikes, hazards, and races. However, the delay model in VHDL might not describe the exact circuit behavior in a real situation, and there are some limitations in the timing model. An improved gate delay model in VHDL is proposed for the correct circuit operation.

The timing model of the VHDL provides the inertial delay and transport delay model [24,26]. In the inertial delay model, the output of a logic function will not respond to input signals stimulating the gate for less than a specified delay time. The transport delay model implies that the output of a logic function will respond faithfully to input signals no matter how briefly the stimulus may be present. In order to represent the circuit behavior close to a real situation, we need an improved model. First, timing constraints, such as set up time and hold time, should be satisfied at the inputs. Second, timing fault at the input signal should be checked and removed. Third, there should be the stability criterion of the inputs. That is, the inputs must remain stable for a given amount of time in order for the outputs to change. Fourth, provided the stability criterion is met, there is a propagation delay associated with the operation of the gate. Finally, the output line is assigned the value of the function of the gate operating on the inputs.

Consider an inverter as an example. At first glance, one might represent this as:

```
B <= not A after T1 ns;
```

However, this fails to differentiate between the stability and propagation delay - requiring inputs to remain stable longer than necessary. A more accurate description using our criterion would be:

```
temp <= not A after T2 ns;
B <= transport temp after T3 ns;
```

where an inertial delay of T2 is used for the stability criterion, and a transport delay of T3 is used for the propagation delay of the inputs. For the inertial delay model, T1 = T2 + T3. However, for the transport delay model T1 = T3, since T2 = 0 and there is no requirement for the minimum width. Here, an input signal A should be a filtered signal from the input block in the improved delay model.

Figure 5.1 shows our improved gate delay model, which consists of an input block and an output block. The input block is *the constraint checker*. The timing constraints for the input are checked for the set up time and the hold time requirements in the constraint checker of the gate. The set up time specifies that the input should be stable for a duration of time prior to the clock transition that strobes data into the gate. The hold time requirement is that the data should be stable for a minimum amount of time after the clock makes its transition. An assertion to check the set up time specification would be written as follow:

```
Assert  not (not CLK'stable and CLK and DATA'stable(S))
report "Set up Time Failure";
```

Thus if the CLK has just made a positive transition, and the data input has not been stable for the previous S time, a set up time failure will be reported. For the hold time test, one could also write the following assertion.

```
Assert not (not CLK'delayed(H)'stable and CLK'delayed(H) and DATA'stable(H))
report "Hold Time Failure";
```

The output block is composed of three sub-blocks; the inertial filter, the stability filter and the propagation delay element. The output block checks timing faults and assign the propagation delay on the signal which is resulted from he logic unit. First, the minimum duration requirement for the signal, i.e. the stability criterion, is checked in the inertial filter of the output block. This concept is based on the new delay model as described in the inverter example, and it checks if there is any spike or a pulse whose duration is less than the stability criterion. This model is consistent with the VHDL model, which is on the output-side delay model [24]. In the output block, another important element is *the stability filter*, which checks the resultant signal from the logic unit whether or not there is any hazards or races. In the stability filter, it detects the abnormal states or timing faults, and prevents that signal from proceeding into the output of the gate. Thus the timing faults can be prohibited in the output. The stability filter is a main concern in this chapter, and it will be described in detail in following sections. Finally, the resultant signal from the stability filter goes to the propagation delay block, and the actual output signal is generated.

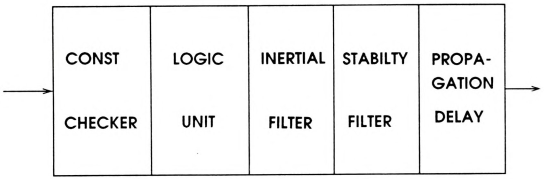

Figure 5.1 : An Improved Gate Delay Model

With this improved delay model in the VHDL, the timing constraints, stability criterion and timing faults can be checked, and the functionally correct circuit operation can be obtained in the logic gate.

In this chapter, timing fault detection methods are described and these procedures could be used as an element for the stability filter in my improved gate delay model.

5.2 Timing Faults in Digital Circuit

When input signals of a logic circuit change their value, we can predict the output signals by looking at its flow table or truth table. If the output signals behave in a different manner, the circuit is said to have a timing fault i.e. a *hazard* or *race* for the input transition [15,18]. Hazards cause unwanted switching transients at the output of a circuit because different paths have different propagation delays. There are four possible hazards, static 0-hazard, static 1-hazard, dynamic hazard and essential hazard. If a circuit output goes to 1 when it should remain 0, it is said that the circuit has a static 0-hazard. And if a circuit output goes to 0 when it should remain 1, it is said that the circuit has a static 1-hazard. The third type of hazard, known as dynamic hazard, causes the output to change three or more times when it should change from 1 to 0 or 0 to 1. The essential hazard is basically a critical race between an input signal change and a feedback signal change. The static 0-hazard and static 1-hazard can be detected by inspecting the diagram of the particular circuit. These two hazards exist because the change of input results in a different product term covering the two minterms [15]. Whenever the circuit must move from one product term to another, there is a possibility of a momentary interval when neither term is equal to 1 (0), giving rise to an undesirable 0 (1) output. These static 0 and 1-hazards can be eliminated by enclosing the two minterms in question with another product term that overlaps both groupings. Thus, these hazards can be eliminated by the addition of redundant gates to the circuit. A physical circuit corresponding to a configuration with a hazard may or may not malfunction, depending upon the magnitude and locations of its delays at a particular time. A hazard-free circuit is one which does not display the type of malfunction regardless of the value of delays [23]. In a combinational circuit, spurious output pulses may not be harmful, depending on how the output is used. However, a hazard becomes very complicated in the sequential circuit, in which, if a momentary incorrect signal, a hazard, is fed back, it may cause the circuit to enter the wrong stable state, thus converting a transient error into a steady-state error [15].

A race condition exists in an asynchronous sequential circuit when two or more state variables change their values in response to a change of an input variable. When unequal delays are encountered, a race condition may cause the state variables to change in an unpredictable manner. There are two possible races, *noncritical* and *critical* races. In a combinational circuit, races just make temporary false output. However, in an asynchronous sequential circuit, they may cause wrong stable states. In noncritical races, the final stable state where the circuit reaches, does not depend on the order in which the state variables change. Noncritical races do not cause any serious problem functionally. In critical races, it is possible to end up in two or more different stable states depending on the order in which the state variables change. Critical races make a circuit work abnormally and they must be eliminated.

5.3 Timing Faults Detection Method

A unified approach to the hazard detection in both combinational and sequential circuits using ternary logic has been reported [3,9]. Eichelberger's hazard detection scheme based on ternary logic [8] is briefly described. A signal 1/2 is used to represent an indeterminate signal which may be either 1 or 0. The ternary function G^*, of a logic gate that realizes the binary function G, can be defined on {0,1,1/2}. A combinational logic circuit is said to contain a *hazard* for an input variable if and only if (1) the output before the change is equal to the output after the change and (2) during the change a spurious pulse may appear on the output. The problem of determining whether or not a hazard in a sequential circuit can be divided into two parts. The first part is to determine all the internal signals that may be changing as a result of the input change, and the second part is to determine whether or not these internal signals will eventually stabilize in some predetermined state. Through these steps, we can determine which of the signals will be unstable as an indeterminate state (1/2) for a result of the changing input variables.

5.4 VHDL Implementation for Timing Faults Detection

In order to get the transient analysis for digital circuit, a hazard condition should be checked on both of the combinational circuits or asynchronous sequential circuits by VHDL. Binary logic of VHDL is extended into multiple valued logic and used for the hazard detection procedures.

The feasible method in making sure for there is no hazard is that various delays in the circuit are not related to lead to the hazard. This can be achieved by inserting adequate delay elements into the path, but in practice the estimates of the various delay values often are difficult and even impossible in complex circuits. Thus we need to apply a simple delay model in order to detect a hazard condition with VHDL. In our procedures, all of the components in a circuit are assumed to have unit delay, or a delta delay. If we do not have this assumption and apply some other delay model, i.e. some length of inertial delay and transport delay, then we cannot check if there is any hazard or not, since the circuit behaviors become different depending on their delay model. For example, in any event, transient false signals can always be filtered out by the action of delay elements with the inertial property. The goal is to find the circuit, which possibly creates a hazard for a particular transition from the design, assuming delta delay model. It is also assumed that for any circuit, all the effects of one input change reach all the outputs before any of the effects of the next change reach any of the outputs. This assumption has essentially the property that the circuit becomes a steady-state condition before any input signal is changed.

A hazard condition can be recognized by the detection method based on a ternary logic, which is implemented using VHDL. The first step is to extend the logical operation in VHDL. In VHDL, the logical operators AND, OR, NOT were defined for predefined types BIT and BOOLEAN, not for ternary logic [25]. In order to implement the detection method based on ternary logic, it is required to extend the logical operation to include the ternary operation. Extension can be done by defining new functions and packages about multiple valued logical operations. One typical example involving multiple-valued AND operation is shown in Figure 5.2. Multiple-valued OR and INV operation can be described in a similar way.

```
package  wfun2 is
   function MVL_AND (DRIVERS : MVL_vector) return MVL ;
   subtype  WABIT        is MVL_AND MVL ;
   type  WABIT_VECTOR    is array (NATURAL range <> ) of WABIT;
end wfun2;

package body wfun2 is
  function MVL_AND (DRIVERS : MVL_vector) return MVL is
    variable COUNT : integer;
    variable VALUE : MVL;
  begin
    VALUE := '0';
    for COUNT in DRIVERS'LOW to DRIVERS'HIGH loop
      if DRIVERS(COUNT) = '0' then VALUE := '0';
          exit;
      elsif DRIVERS(COUNT) = 'R' then VALUE := 'R' ;
      elsif DRIVERS(COUNT) = '1' then
          if VALUE := 'R' then VALUE := 'R';
          else VALUE := '1';
          end if;
      end if;
    end loop;
    return VALUE;
  end MVL_AND;
end wfun2;
```

Figure 5.2 A Multiple Valued AND function

In above function, the indeterminate state 1/2 is set as "R".

The following procedure detects the hazard condition in combinational circuits.

Procedure 1 - Hazard Detection in combinational circuits for specific transition

1) Change the input value as following sequences

   A -> R (1/2) -> B

2) Find out the output value

   OUT(A) = OUT(B) ≠ R and OUT(R) = R

3) If all of two conditions in 2) are satisfied, this circuit contains the hazard.

Here, A is the initial input state and B is the final state respectively. In this procedure, we can detect the hazard regardless of the gate delay of the logic circuits. To test the above condition (2), we need the following process.

```
process
    begin
        wait until (not OUT'stable)
        Qu1 <= OUT ;
        wait until (not OUT'stable)
        Qu2 <= OUT ;
        wait until (not OUT'stable)
        Qu3 <= OUT ;
        Assert (not ( (Qu1 = Qu3) and not (Qu1 = 'R') and Qu2 = 'R'))
        report " Hazard Condition Detect " ;
end process;
```

In the above model, each output value OUT will be assigned to Qu1, Qu2, and Q3 whenever its value is changed. Assertion statement executes after the last transaction occurs from OUT to Qu3.

For sequential circuits, we can similarly define the procedure which detects whether or not a circuit contains hazards for a particular input change.

Procedure 2 - Hazard Detection in sequential circuits for specific transition

1) Change the input value as following sequences

   A -> R (1/2) -> B

2) Find the output value

3) If the output value is stable and R (1/2), this circuit contains the hazard.

   The output signal is unstable and may have several intermediate

   state during the transition.

Selected example circuits have been tested for hazard detection using above procedures, and VHDL simulation results show our procedures are working. There are three examples which have been tried, which are 2 static hazard circuits and 1 hazard-free circuit. One of those examples is shown in Figure 5.3, and a static hazard appears in this combinational circuit. A main portion of a VHDL program and its simulation result is attached. The whole program lists and two other examples are shown in Appendix 2.

```
architecture static1m of hazard1m  is
  begin
    B: block
        signal S1, S2, S3 : MVL := '0' ;
        for L_inv: C_inv use entity inv_gate(mul_inv);
        for L_and1, L_and2: C_and use entity and_gate(mvl_and);
        for L_or : C_or  use entity  or_gate(mul_or);
        begin
           L_and1: C_and port map (X1, X2, S2);
           L_inv: C_inv port map (X2, S1);
           L_and2: C_and port map (S1, X3, S3);
           L_or : C_or  port map (S2, S3, Yout);
        end block;
end static1m;

architecture arc_hazard1m of test  is
  begin
    B: block
        component C_hazard1m
           port (X1, X2, X3 : in MVL;
                  Yout       : out MVL);
        end component;
        signal X1, X2, X3 : MVL := '0' ;
        signal Yout : MVL := '0' ;
        signal Qu1, Qu2, Qu3 : MVL := '0' ;
        for all: C_hazard1m use entity hazard1m(static1m);
        begin
           X1 <= '1' after 0ns ;
           X2 <= '1' after 0ns, 'R' after 50ns, '0' after 100ns ;
           X3 <= '1' after 0ns ;
           L_hazard1m: C_hazard1m port map (X1, X2, X3, Yout);

        -- Hazard Detection Assert Statement
           process
             begin
                wait until (not Yout'stable);
                Qu1 <= Yout ;
                wait until (not Yout'stable);
                Qu2 <= Yout ;
                wait until (not Yout'stable);
                Qu3 <= Yout ;
                Assert (not ( (Qu1 = Qu3) and
                              not (Qu1 = 'R') and Qu2 = 'R'))
                report " Hazard Condition Detect " ;
                wait until (true=false) ;
             end process;

        end block;
end arc_hazard1m;
```

```
     TIME    |---------------------------SIGNAL NAMES---------------------------|
             |
     (FS)    |     X        X        X        Y        S        S        S
             |     1        2        3        O        1        2        3
             |                                U
             |                                T
             |
           0 |    '0'      '0'      '0'      '0'      '0'      '0'      '0'
          +1 |    '1'      '1'      '1'      ***      ***      ***      ***
      500000 |    ***      ***      ***      ***      ***      '1'      ***
     1000000 |    ***      ***      ***      '1'      ***      ***      ***
    50000000 |    ***      'R'      ***      ***      ***      ***      ***
    50500000 |    ***      ***      ***      ***      'R'      'R'      ***
    51000000 |    ***      ***      ***      'R'      ***      ***      'R'
   100000000 |    ***      '0'      ***      ***      ***      ***      ***
   100500000 |    ***      ***      ***      ***      '1'      '0'      ***
   101000000 |    ***      ***      ***      ***      ***      ***      '1'
   101500000 |    ***      ***      ***      '1'      ***      ***      ***
```

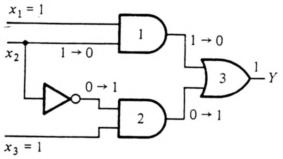

Figure 5.3 Static hazard in combinational circuit

Chapter 6. Conclusion and Future Works

The VLSI system can be modeled and simulated for its verification using VHDL. In this thesis, some useful modeling techniques have been presented for the efficient modeling of the VLSI systems. VHDL semantics on process statements and functions are described and the delay characteristics are discussed. A new method for design verification has been proposed in this paper. Using timing tolerance information supplied by the design, it can be verified that the structural description is equivalent to the behavioral model. This technique is used to verify both combinational and sequential circuits. A simple type of timing faults is considered in the verification. In order to verify VHDL models efficiently, an additional feature is necessary in VHDL . To model gate delay precisely, an improved gate delay model is required. In order to detect timing faults, such as hazards and races, a detection procedure based on ternary logic has been proposed. The automatic detection procedure for timing faults with VHDL has been implemented, and hazards can be detected with this procedure.

Further research is suggested to improve the results. First, the proposed hazard detection procedure assumes a delta delay on all gate components in the circuit. In a real circuit, however, all of the gate components have some amount of delay length. A general transient analysis algorithm, which is based on a real delay model in the circuit, should be developed to be used for timing analysis of the digital circuits. Second, if a *VHDL reasoning system*, which reasons about circuit behavior, is available, then this reasoning system could be used to analyze timing behavior and detect timing faults [9,16,17]. Such a reasoning system would be a great help in design verification and in transient analysis for digital circuits.

BIBLIOGRAPHY

Bibliography

1. Amstrong, J., Chip Level Modeling with VHDL, Prentice Hall, 1989.
2. Augustin L.M. et al, "Verification of VHDL Designs Using VAL," Proceedings of Design Automation Conference 1988, pp. 48-53.
3. Breuer, M and A. Friedman, Diagnosis and Reliable Design of Digital Designs, Rockville, MD, Computer Science Press, 1976.
4. Breuer, M, "A Note on Three-Valued Logic Simulation," IEEE Transactions on Computer, April 1974, pp.
5. Bryant, R. E., "Race Detection in MOS Circuits by Ternary Simulation," VLSI'83, pp. 85-95.
6. Chung, M.J., Jin-Hyung Lee, C.Y. Lee, B. Reidenbach, "VHDL Modeling Benchmark Test: The Radio SINCGARS Circuitry," Proceedings of Government Microelectronics Conference (GOMAC), Olrando, FA., Nov. 1989.
7. Coelho, D.R., VHDL Handbook, Kluwer Academic Press, 1989.
8. Eichelberger, E. B., "Hazard Detection in combinational and sequential switching circuits, " IBM Journal of R&D, vol.9, March 1965, pp. 90-99.
9. Fujita, M, H. Tanaka, T. Moto-Oka, "Logic Design Assistant with Temporal Logic," Proceedings of Computer Hardware Description Languages, 1985, pp. 129-139.
10. Lee, Jin-Hyung and Moon Jung Chung, "Design Verification with VHDL," Conference Record of 1990 Spring VHDL Meeting, Boston, MA., April 1990.

11. Lee, Jin-Hyung, M.J. Chung, C.Y. Lee, B. Reidenbach, "VHDL Modeling Benchmark Test," Conference Record of 1989 Spring VHDL Meeting, Washington D.C., June 1989.

12. Leung, S. and M. Shanblatt, ASIC Design with VHDL, Kluwer Academic Press, 1989.

13. Liaw, H, K. Tran and C Lin, "VVDS: A Verification/Diagnosis System for VHDL," Proceedings of Design Automation Conference - 1989, pp. 435-440.

14. Lightner, Michael R, "Modeling and Simulation of VLSI Digital Systems", Proceedings of the IEEE, Vol. 75, No.6, June 1987, pp.786-796.

15. Mano, M, Digital Design, Chapter 9, Prentice-Hall, 1985.

16. Maruyama, F. and M. Fujita, "Hardware Verification," IEEE Computer, Feb. 1985, pp. 22-32.

17. Moszkowski, B., "A Temporal Logic for Multi-Level Reasoning about Hardware," Proceedings of Computer Hardware Description Languages, 1983, pp. 79-89.

18. Hill, F. J. and G. R. Peterson, Introduction to Switching Theory and Logical Design, Wiley-Sons, 1981.

19. Huh, Y et al, "Semantics of VHDL Designs using VAL," Proceedings of ICCD-86, Dec. 1986, pp.10-14.

20. Shahdad, M., R. Lipsett, E. Marschner et al, "VHSIC Hardware Description Language," IEEE Computer, Feb. 1985. pp. 94-102.

21. Ohkura, I, et al, "VLSI Design Verification and Logic Simulation", Hardware and Software Concept in VLSI, Van Nostrand Reinhold, Ed. Guy Rabbat, 1983, pp. 524-551.

22. Unger, S. H., "Hazards and Delays in Asynchronous Sequential Switching Circuits," IRE transactions on Circuit Theory, March 1959, pp.12-25.

23. Unger, S. H., Asynchronous Sequential Switching Circuit, Wiley Interscience, 1969.

24. Special Issue on VHSIC Hardware Description Language, IEEE Design & Test, April 1986.

25. "IEEE Standard VHDL Language Reference Manual," IEEE std 1076-1987, IEEE Inc, March 31, 1988.

26. "VHDL Tutorials for IEEE Standard 1076 VHDL," Second Draft, CAD Language Systems Inc., June 1987.

27. "VHDL User's Manual, Volume 1 : Tutorial," Intermetrics Inc., April 1987.

28. "User's Manual for the Standard VHDL 1076 Support Environment", Sun/3 Version, Intermetrics Inc., Nov. 1988.

APPENDICES

APPENDIX A:

VHDL Modeling for Bit Sync Filter

Appendix A. VHDL Modeling for Bit Sync Filter

1. Bit Sync Filter Circuit Description

The Bit Sync Filter will synchronize a 20 khz or 16 khz clock to a data stream by inserting or deleting clock pulses on a 320 khz clock. The circuit adjusts this clock by the in-phase clock time (falling edge of the clock) and at the mid-phase clock time (rising edge of the time). If the data transition occurs before the clock edge, extra clock pulses will be added in order to advance the clock edge to where the transitions are occuring. Similarily, if the data transition occurs after the clock edge, pulses will be deleted in order to delay the clock to the transitions. The circuit also functions in two modes: a coarse mode that adjusts the clock immediately, and a fine mode that reduces clock jitter by requiring a number of transitions on the same side of the clock edge before an adjustment is made. Figure A.1 shows the block diagram of the Bit Sync Filter.

Two major blocks in this chip are the clock generation block and phase detection/adjust block. The clock generation block consists of a programmable divide-by-1/2/3 counter and three other counters that generates two symmetrical clocks at 20 khz and 16 khz. The programmable counter has two inputs, labeled P1 and P0, that control the divide by rate. The divider works as a pulse swallower that in the normal divide-by-two mode (P1 = 0, P0 = 1) inhibits every other clock pulse from appearing on the output. When the clock is to be advanced, the divider

will be placed in the divide-by-one mode (P0 = 0, P1 = 0) for one cycle and the clock pulse will not be swallowed; when the clock is to be delayed, the divider will be placed in the divide-by-three mode (P1 = 1, P0 = 0) for one cycle and two clock pulses will be swallowed.

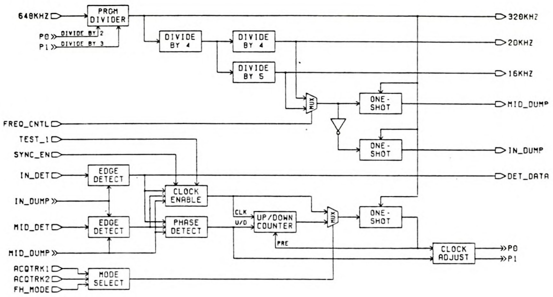

Figure A.1 Block Diagram of the Bit Sync Filter

The phase detection block consists of two edge detectors, up/down counter, clock adjustment circuit and some random logics. The in-phase and mid-phase data from an external interface and dump circuit are routed to two edge detectors. The in-phase data signal is examined for a change in state between two successive in-phase clock edges in the in-phase edge detector. For the mid-phase edge detector, the in-phase and the mid-phase data signal are examined for a change between the mid-phase data signal sampled at mid-phase clock edge and in-phase data signal at the next in-phase clock edge. If both edge detects are true, it indicates that the clock is leading the data; if the mid-phase edge detect is false while the in-phase edge detect is true, it indicates that the clock is trailing the data. The up/down signal is determined by the state of both edge detect circuits. When the phase detect signal (up/down) indicates that the clock leads the data, P1 is set high while P0 is set to low. When the phase detect signal indicates that the clock lags the data, both P1 and P0 will be set low.

2. VHDL Modeling of Bit Sync Filter

2.1 Functional Block Decomposition

The basic unit in VHDL modeling is a design entity, which can be of any complexity, from a simple logic gate to a whole Bit Sync Filter chip. The design entity is composed of two parts: an entity declaration and an architectural body. An entity declaration describes the interface, such as input and output ports and generic parameters, of a design entity. An architectural body represents a distinct view of a design entity. It can be a behavioral or a or structural description of an design entity. To model the bit sync filter, it has been partitioned into 16 functional blocks. In addition to these block, two clock generation blocks and a bus resolution function block is needed. Each block, called a component, represents an entity which performs a well encapsulated function. Each component has been modeled at the level of behavioral description then merged to a top level model of bit sync filter. Some critical blocks for the timing has been modeled and verified at the level of structural description or mixed description of both for the purpose of the precise modeling. In this section, some modeling techniques for major components, such as phase detector, programmable divider, up/down counter, clock adjustment, edge detector and multiplexer are described.

2.2 Components Modeling

1. Edge Detector

A clock, selected by a freq_cntl signal, is routed to two edge detectors. The in-dump and mid-dump signals detect the falling edge and the rising edge of the selected clock respectively. In order to model edge detector, two process statements are used. One process is for initialization and the other one is for edge detection. The in_dump signal goes to 1 at sel_clk = '0', and go back to '0' at next clock signal. The mid_dump signal goes to '1' and back to '0' at next clock signal.

2. Phase Detector

In the phase detector logic, an in-phase detection block checks for data transition in a clock cycle and the output result is generated in the following clock cycle. A mid-phase detection block checks for data transition in the first half of clock cycle and the result is synchronized with in-phase detection result. If a data transition occurs between two in-dump signals, then the in-phase signal goes to 1 at the following in-dump signal and goes back to 0 at the next in-dump signal. For the mid_phase signal, it checks if there is a change between mid_phase data signal (MID_DET) sampled at mid_dump and in _phase data signal (in_det) sampled at the next in-dump signal. For modeling of in_phase edge detector, two process statements and some wait statements are used. One process is for initialization of phase detect signal and the other process is for main module of phase detection. In in_phase edge detector model, it checks if two in-phase data signal sampled at in_dump signal and following in_dump signal is equivalent or not. If two signals are same, then in_phase signal goes to '1'. Otherwise, it goes to '0'. This operation can be implemented by the wait statement in a process. The mid_phase phase detector has been implemented in a similar way.

3. Programmable Divider and Clock Divider

A programmable divider operates in three modes. In the normal divide-by-2 mode, it works as a pulse swallower and allows every other clock pulse. When the clock is to be advanced, the divider will be placed in divide-by-1 mode and the clock pulse will not be swallowed. When the clock is to be delayed, the divider will be placed in divide-by-3 mode and two clock pulses will be swallowed. In this VHDL model, the count is a signal for counting pulses. P0 and P1 are the control input for the divide-by rate and K640khz is an input clock signal. In divide-by-2 mode, whenever input signal make a 0 to 1 transition the value of count is increased. If this block counts twice, then count is set to 0 and an output pulse is generated. Divide-by-3 mode is quite similar to divide-by-2 mode.

The clock dividers, which generates divide-by-16 and divide-by-20 signal from the combination of the divide-by-4 and divide-by-5, has been implemented in a similar way.

4. Up / Down Counter

In the tracking mode, a clock adjustment is made only when a majority of 30 transitions to one side or the other of the active clock edge is detected. When the up/down signal is low, the counter will count up by one with each clock. When the signal is high, the counter will count down. When the up/down counter counts 30 transitions, then an output pulse is generated to change the divide by rate and also preset the counter back to initial state. But setting the preset value is not simple and depends on the value of the enable signal (PE). When PE is high, the clock is disabled and the signals out0 and out62 are held low. The count will be held at its current value and will be loaded with a new count on the first rising edge of the clock after PE goes back low. The value that is loaded depends on two conditions: the counter value when PE went high, and the state on up/down signal when this first clock occurs after PE goes back low. The new counter value is determined as follows:

if the count was even and up/down signal is low, the count is set to 33.

if the count was odd and up/down signal is low, the count is set to 32.

if the count was even and up/down signal is high, the count is set to 31.

if the count was odd and up/down signal is high, the count is set to 30.

VHDL model consists of two processes, in which one is for deciding the enable signal and the other is for counting the clock pulses.

5. Multiplexer

The multiplexer selects the data clock rate to which the circuit is to be synchronized. It selects 20khz when FREQ_CNTL = 0, and 16khz, otherwise. We can implement the multiplexer in VHDL in several ways, as described in chapter 2. One implementation is by using the *when statement*. Another implementation is using a bus resolution function of the WIRED_OR function. In the bus resolution function, signal S is a guarded signal of a register, and is multiplexed from a driver of the block whose guard expression is true.

6. Clock adjustment block

This block controls the programmable divider signals P0 and P1. When the final_shot is triggered either by the clock enable or the up/down counter, a clock adjustment signal is generated to change the divide by rate and also preset the up/down counter back to initial state. When the up/down signal is '0', P1 is set high while P0 is set low. When the up/down signal is '1', both P1 and P0 signals will be set low. When the final_shot is not active, P1 is set low while P0 is set high.

2.3 Description of VHDL Function blocks

divider_prgm : programmable divider block in the clock generation section

divider_16 and divide_20 : divider by 16 and 20 circuit

mux1 : select K20khz or K16khz depends on freq_cntl signal

in_shot : generates in_dump signal for selected clock

mid_shot : generates mid_dump signal for selected clock

in_edge_det : in_phase detection block

mid_edge_det : mid_phase detection block

clk_enable : clock enable signal generation block

phase_detect : deciding phase depending on in_phase and mid_phase

up_down : up/down counter block

mode_select : acquisition mode or tracking mode selection

mux2 : generate signal for clock adjustment

final_shot : one-shot from clock enable or up/down counter

clk_adjust : clock adjustment block and generate P1 and P0 signal

period_clock : input clock generator block

period_testclock : test_1 signal generator block

Appendix B:

Examples for Transient Analysis with VHDL

3. Conclusion

The modeling of bit sync in VHDL/1076 has been carried out successfully. The behavior of each functional block of bit has been modeled in VHDL. For the top model of bit sync filter, it has been modeled by using its partitioned components. For the purpose of verification, some important blocks have been modeled in the level of structural or mixed description and they are compared to behavioral models. VHDL codes for this project can be used as a specification of bit sync filter circuit. For each functional block, the design can be carried out by the designer himself, or can be drawn from an existing cell library. Several alternatives of the design can exist as long as it is consistent to its behavior. The VHDL simulation result was compared with simulation results from the Mentor Graphics circuit simulator. Two simulation results are exactly matching each other.

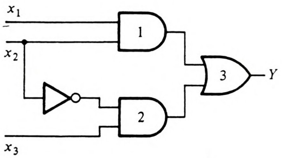

Figure B.1 : A Circuit for Static Hazard in Combinational Circuit

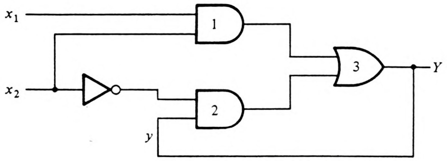

Figure B.2 : A Circuit for Static Hazard in Sequential Circuit

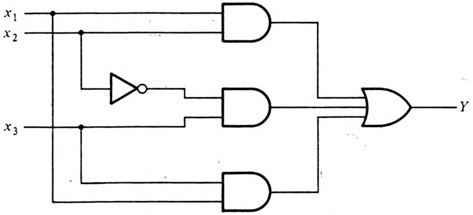

Figure B.3 : A Hazard Free Circuit

```
--  VHDL Program for Static Hazard in combinational Circuit --

                    Example Circuit: Figure B.1
```

```
library Work;
use Work.all;
use defs.all;
use wfun1.all;
use wfun2.all;
use wfun3.all;


entity hazard1m is
port ( X1, X2, X3 : in  MVL ;
            Yout   : out MVL );
end hazard1m;

architecture static1m of hazard1m  is
  begin
    B: block
        component C_inv
              port (In1 : in MVL;
                    output : out MVL);
        end component;
        component C_and
              port (And_In1, And_In2 : in MVL;
                    output   : out MVL);
        end component;
        component C_or
              port (In1, In2 : in MVL;
                    output   : out MVL);
        end component;

        signal S1, S2, S3 : MVL := '0' ;

        for L_inv: C_inv use entity inv_gate(mul_inv);
        for L_and1, L_and2: C_and use entity and_gate(mvl_and);
        for L_or : C_or  use entity  or_gate(mul_or);

        begin

           L_and1: C_and port map (X1, X2, S2);
           L_inv: C_inv port map (X2, S1);
           L_and2: C_and port map (S1, X3, S3);
           L_or : C_or  port map (S2, S3, Yout);

        end block;
end static1m;
```

```
entity test is end test;

architecture arc_hazard1m of test  is
  begin
    B: block

        component C_hazard1m
           port (X1, X2, X3 : in MVL;
                  Yout       : out MVL);
        end component;

        signal X1, X2, X3 : MVL := '0' ;
        signal Yout : MVL := '0' ;
        signal Qu1, Qu2, Qu3 : MVL := '0' ;

        for all: C_hazard1m use entity hazard1m(static1m);

        begin
           X1 <= '1' after 0ns ;
           X2 <= '1' after 0ns, 'R' after 50ns, '0' after 100ns ;
           X3 <= '1' after 0ns ;

           L_hazard1m: C_hazard1m port map (X1, X2, X3, Yout);


        -- Hazard Detection Assert Statement

           process
             begin
                wait until (not Yout'stable);
                Qu1 <= Yout ;
                wait until (not Yout'stable);
                Qu2 <= Yout ;
                wait until (not Yout'stable);
                Qu3 <= Yout ;
                Assert (not ( (Qu1 = Qu3) and
                              not (Qu1 = 'R') and Qu2 = 'R'))
                report " Hazard Condition Detect " ;
                wait until (true=false) ;
             end process;

        end block;
end arc_hazard1m;
```

```
                         Vhdl Simulation Report

        Report Name: Report of Static Hazard in Combinational Ckt"
               Kernel Library Name: <<LEEJIN>>ARC_HAZARD1M
                    Kernel Creation Date: MAY-18-1989
                      Kernel Creation Time: 19:51:30
                             Run Identifer: 1
                          Run Date: MAY-18-1989
                            Run Time: 19:51:30

               Report Control Language File: shaz_com1m.rcl
                  Report Output File : ARC_HAZARD1M.rpt

                      Max Time: 9223372036854775807
                          Max Delta: 2147483646

Report Control Language :

Simulation_report simulation_1 is
    begin
        Report_name is "Report of Static Hazard in Combinational Ckt";
        Page_width is 80;
        Page_length is 50;
        Remove_page_id;
        Signal_format is horizontal;
        Select_start_time 0 ns;
        Select_stop_time 1000 ns;
        Sample_signals by_event using '*';
        Select_Signal .B: X1, X2, X3, Yout ;
        Select_Signal .B/L_hazard1m.B: S1, S2, S3 ;
    end simulation_1;

Report Format Information :

 Time is in FS relative to the start of simulation
 Time period for report is from 0 FS to 1000000000 FS
 Signal values are reported by event  ( '*' indicates no event )


 TIME      |----------------------------SIGNAL NAMES---------------------------|
           |
 (FS)      |       X         X         X         Y         S         S         S
           |       1         2         3         O         1         2         3
           |                                     U
           |                                     T
           |
        0  |      '0'       '0'       '0'       '0'       '0'       '0'       '0'
       +1  |      '1'       '1'       '1'       ***       ***       ***       ***
   500000  |      ***       ***       ***       ***       ***       '1'       ***
  1000000  |      ***       ***       ***       '1'       ***       ***       ***
 50000000  |      ***       'R'       ***       ***       ***       ***       ***
 50500000  |      ***       ***       ***       ***       'R'       'R'       ***
 51000000  |      ***       ***       ***       'R'       ***       ***       'R'
100000000  |      ***       '0'       ***       ***       ***       ***       ***
100500000  |      ***       ***       ***       ***       '1'       '0'       ***
101000000  |      ***       ***       ***       ***       ***       ***       '1'
101500000  |      ***       ***       ***       '1'       ***       ***       ***
```

-- VHDL Program for Static Hazard in Sequential Circuit --

Example Circuit : Figure B.2

```
library Work;
use Work.all;
use Defs.all;
use wfun1.all;
use wfun2.all;
use wfun3.all;


entity hazard2m is
     port ( X1, X2 : in MVL ;
            Yout   : out MVL );
end hazard2m;

architecture sequenm of hazard2m  is
  begin
    B: block
        component C_inv
            port (In1 : in MVL;
                  output : out MVL);
        end component;
        component C_and
            port (And_In1, And_In2 : in MVL;
                  output   : out MVL);
        end component;
        component C_or
            port (In1, In2 : in MVL;
                  output   : out MVL);
        end component;

        signal S1, S2, S3, SS : MVL := '0' ;

        for L_inv: C_inv use entity inv_gate(mul_inv);
        for L_and1, L_and2: C_and use entity and_gate(mvl_and);
        for L_or : C_or  use entity  or_gate(mul_or);

        begin

        L_and1: C_and port map (X1, X2, S2);
        L_inv: C_inv port map (X2, S1);
        L_and2: C_and port map (S1, SS, S3);
        L_or : C_or  port map (S2, S3, SS);

        Yout <= SS  after 0.5ns;

   end block;
end sequenm;
```

```
entity test is end test;

architecture arc_hazard2m of test  is
  begin
   B: block

        component C_hazard2m
            port (X1, X2 : in MVL;
                  Yout   : out MVL);
        end component;

        signal X1, X2 : MVL := '0' ;
        signal Yout : MVL := '0' ;

        for all: C_hazard2m use entity hazard2m(sequenm);

        begin
           X1 <= '1' after 0ns ;
           X2 <= '1' after 0ns, 'R' after 50ns, '0' after 100ns;

           L_hazard2m: C_hazard2m port map (X1, X2, Yout);


        -- Hazard Detection Assert Statement

           process
             begin
               wait for 100ns;
               Assert  not ( Yout'stable and Yout = 'R' )
               report " Hazard Condition Detect " ;
               wait until (true=false);
           end process;

     end block;
end arc_hazard2m;
```

```
                          Vhdl Simulation Report

        Report Name: Report of Static Hazard in Sequential Circuit"
                 Kernel Library Name: <<LEEJIN>>ARC_HAZARD2
                     Kernel Creation Date: MAY-17-1989
                       Kernel Creation Time: 16:13:36
                              Run Identifer: 1
                          Run Date: MAY-17-1989
                            Run Time: 16:13:36

                 Report Control Language File: shaz_seq.rcl
                    Report Output File : ARC_HAZARD2.rpt

                       Max Time: 9223372036854775807
                           Max Delta: 2147483646

 Report Control Language :

 Simulation_report simulation_1 is
     begin
         Report_name is "Report of Static Hazard in Sequential Circuit";
         Page_width is 80;
         Page_length is 50;
         Remove_page_id;
         Signal_format is horizontal;
         Select_start_time 0 ns;
         Select_stop_time 1000 ns;
         Sample_signals by_event using '*';
         Select_Signal .B: X1, X2, Yout ;
         Select_Signal .B/L_hazard2.B: S1, S2, S3, SS ;
     end simulation_1;

 Report Format Information :

  Time is in FS relative to the start of simulation
  Time period for report is from 0 FS to 1000000000 FS
  Signal values are reported by event   ( '*' indicates no event )


  TIME     |---------------------------SIGNAL NAMES---------------------------|
           |
  (FS)     |        X        X        Y        S        S        S        S
           |        1        2        O        1        2        3        S
           |                          U
           |                          T
           |
        0  |       '0'      '0'      '0'      '0'      '0'      '0'      '0'
       +1  |       '1'      '1'      ***      ***      ***      ***      ***
   500000  |       ***      ***      ***      ***      '1'      ***      ***
  1000000  |       ***      ***      ***      ***      ***      ***      '1'
  1500000  |       ***      ***      '1'      ***      ***      ***      ***
 50000000  |       ***      'R'      ***      ***      ***      ***      ***
 50500000  |       ***      ***      ***      'R'      'R'      ***      ***
 51000000  |       ***      ***      ***      ***      ***      'R'      'R'
 51500000  |       ***      ***      'R'      ***      ***      ***      ***
100000000  |       ***      '0'      ***      ***      ***      ***      ***
100500000  |       ***      ***      ***      '1'      '0'      ***      ***
```

-- VHDL Program for Hazard Free Circuit --

Example Circuit : Figure B.3

```
library Work;
 use Work.all;
 use defs.all;
 use wfun1.all;
 use wfun2.all;
 use wfun3.all;


entity hazard3m is
    port ( X1, X2, X3 : in  MVL ;
           Yout       : out MVL );
end hazard3m;

architecture free1m of hazard3m  is
  begin
   B: block
        component C_inv
           port (In1 : in MVL;
                 output : out MVL);
        end component;
        component C_and
           port (And_In1, And_In2 : in MVL;
                 output   : out MVL);
        end component;
        component C_or2
           port (In1, In2 : in MVL;
                 output   : out MVL);
        end component;
        component C_or3
           port (In1, In2, In3 : in MVL;
                 output   : out MVL);
        end component;

        signal S1, S2, S3, S4 : MVL := '0' ;

        for L_inv: C_inv use entity inv_gate(mul_inv);
        for L_and1, L_and2: C_and use entity and_gate(mvl_and);
        for L_or1: C_or2 use entity or_gate(mul_or);
        for L_or2: C_or3 use entity or_gate3(mul_or3);

        begin

          L_and1: C_and port map (X1, X2, S2);
          L_inv: C_inv port map (X2, S1);
          L_and2: C_and port map (S1, X3, S3);
          L_or1: C_or2 port map (X1, X3, S4);
          L_or2: C_or3 port map (S2, S3, S4, Yout);

      end block;
end free1m;
```

```
entity test is end test;

architecture arc_hazfreem of test  is
   begin
     B: block

         component C_hazard3m
              port (X1, X2, X3 : in MVL;
                    Yout       : out MVL);
         end component;

         signal X1, X2, X3 : MVL := '0' ;
         signal Yout : MVL := '0' ;
         signal Qu1, Qu2, Qu3 : MVL := '0' ;

         for all: C_hazard3m use entity hazard3m(free1m);

         begin
            X1 <= '1' after 0ns ;
            X2 <= '1' after 0ns, 'R' after 50ns, '0' after 100ns ;
            X3 <= '1' after 0ns ;

            L_hazard1m: C_hazard3m port map (X1, X2, X3, Yout);


         -- Hazard Detection Assert Statement

            process
               begin
                 wait until (not Yout'stable);
                 Qu1 <= Yout ;
                 wait until (not Yout'stable);
                 Qu2 <= Yout ;
                 wait until (not Yout'stable);
                 Qu3 <= Yout ;
                 Assert (not ( (Qu1 = Qu3) and
                               not (Qu1 = 'R') and Qu2 = 'R'))
                 report " Hazard Condition Detect " ;
                 wait until (true=false) ;
           end process;

       end block;
end arc_hazfreem;
```

```
                    Vhdl Simulation Report

        Report Name: Report of Static Hazard in Combinational Ckt"
               Kernel Library Name: <<LEEJIN>>ARC_HAZFREEM
                    Kernel Creation Date: OCT-15-1989
                      Kernel Creation Time: 18:12:12
                             Run Identifer: 1
                          Run Date: OCT-15-1989
                            Run Time: 18:12:12

               Report Control Language File: shaz_com1m.rcl
                  Report Output File : arc_hazfreem.rpt

                      Max Time: 9223372036854775807
                          Max Delta: 2147483646

Report Control Language :

Simulation_report simulation_1 is
    begin
        Report_name is "Report of Static Hazard in Combinational Ckt";
        Page_width is 80;
        Page_length is 50;
        Remove_page_id;
        Signal_format is horizontal;
        Select_start_time 0 ns;
        Select_stop_time 1000 ns;
        Sample_signals by_event using '*';
        Select_Signal .B: X1, X2, X3, Yout ;
        Select_Signal .B/L_hazard1m.B: S1, S2, S3 ;
    end simulation_1;

Report Format Information :

 Time is in FS relative to the start of simulation
 Time period for report is from 0 FS to 1000000000 FS
 Signal values are reported by event  ( '*' indicates no event )


 TIME      |---------------------------SIGNAL NAMES-------------------------|
           |
 (FS)      |         X         X         X         Y         S         S         S
           |         1         2         3         O         1         2         3
           |                                       U
           |                                       T
           |
         0 |        '0'       '0'       '0'       '0'       '0'       '0'       '0'
        +1 |        '1'       '1'       '1'       ***       ***       ***       ***
    500000 |        ***       ***       ***       '1'       '1'       ***       ***
  50000000 |        ***       'R'       ***       ***       ***       ***       ***
 100000000 |        ***       '0'       ***       ***       ***       ***       ***
```